CS 211: Intro to Computer Architecture

12.1: Single-Cycle RV64I CPU - Arithmetic/Logic

Minesh Patel

Spring 2025 – Tuesday 15 April

Announcements

- Ongoing
  - **WA8:** due **Friday (April 18) @ 11:59 pm**
  - **PA4:** due **Friday (April 18) @ 11:59 pm**

- Upcoming
  - **PA5:** TBA: planned for Friday
  - **WA9:** TBA: planned for Friday

PA6 and WA11

- Turns out both are too ambitious
- We will reweight their percentages

- **Programming Assignments (53%)**: We will have 6 programming assignments weighted as follows.
  - 5% ~~[3%]~~ PA1: Intro to Linux
  - 12% ~~[10%]~~ PA2: C Development + Integers
  - 12% ~~[10%]~~ PA3: Pointers, Arrays, Strings, Structs and Explicit Memory Management
  - 12% ~~[10%]~~ PA4: Assembly Programming
  - 12% ~~[10%]~~ PA5: Single-cycle CPU Emulation
  - ~~[10%] PA6: Extension with Memory + Caches~~

- WAs still worth 12% total (~1.7% each over 7 WAs + 3 dropped)

Layers of Abstraction for CS 211

Real-world Demands

Problem

Algorithm

Program

CS 415: Compilers

Runtime

CS 416: Operating Systems

System software

HW/SW Interface (ISA, ABI)

Microarchitecture

Logic gates

Circuits

Technology

Physics

Real-world Constraints

High-Level Language Program

```
int i;
for(i = 0; i < 10; i++)
    a[i] = b[i] * c[i];
```

Assembly Language Program

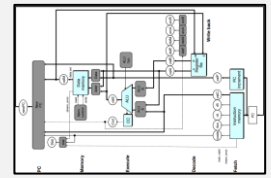
```
loop:  ld   a4, 0(a5)
       ld   a3, 8(a5)
       addi a5, a5,  -8
       add  a4, a4,  a3
       sd   a4, 8(a5)
       bne  a0, a5,  loop
```

Machine Language Program

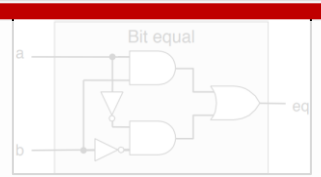
```
0000_0000_0000_0111_1011_0111_0000_0011
0000_0000_1000_0111_1011_0110_1000_0011
1111_1111_1000_0111_1000_0111_1001_0011
0000_0000_1101_0111_0000_0111_0011_0011
0000_0000_1110_0111_1011_0100_0010_0011
1111_1110_1111_0101_0001_0110_1110_0011
```

Instruction Set Architecture (ISA)

Simple RISC-V Processor



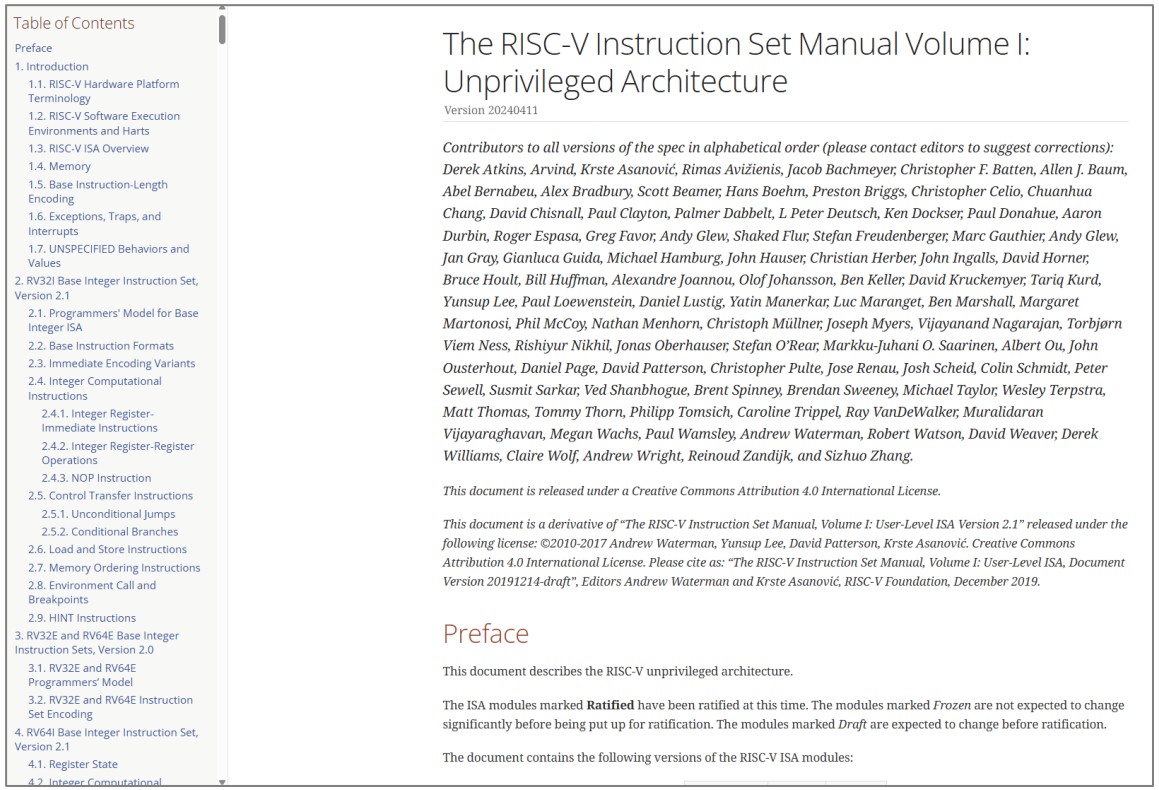
Logic Circuits



Reference Material

- Today's lecture partially draws inspiration from:
  - CS 61C @ UC Berkeley (Prof. Dan Garcia)

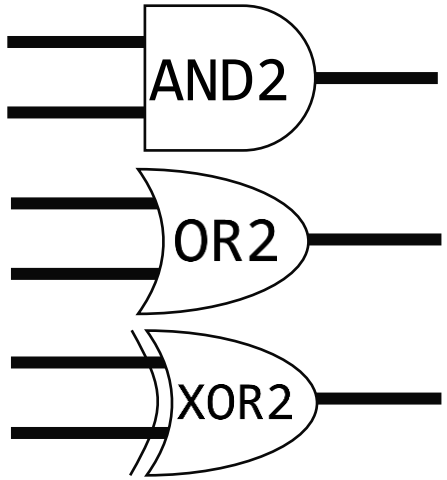
And the RISC-V ISA Manual



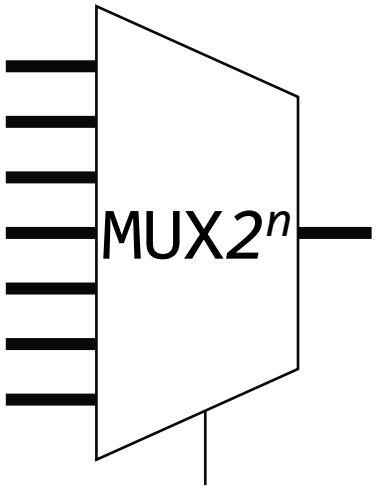
RISC-V ISA Manual (Online Version)

Recap: Combinational Logic Circuits

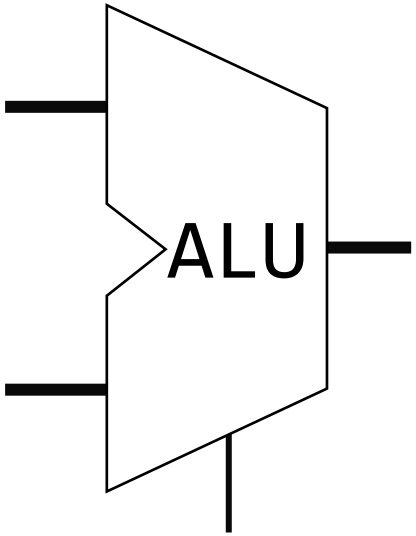
Individual Logic Gates



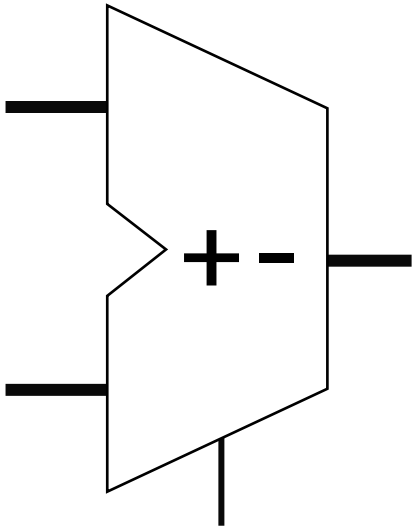
Multiplexer



ALU



Add/Sub



Today: Combine these to execute a RISC-V program

Agenda

- **Single-Cycle CPU**
  - State Elements
- Five-Stage Execution Model
  - Implementing an ADD
  - R-TYPE Instructions
  - Any Instruction

Goal

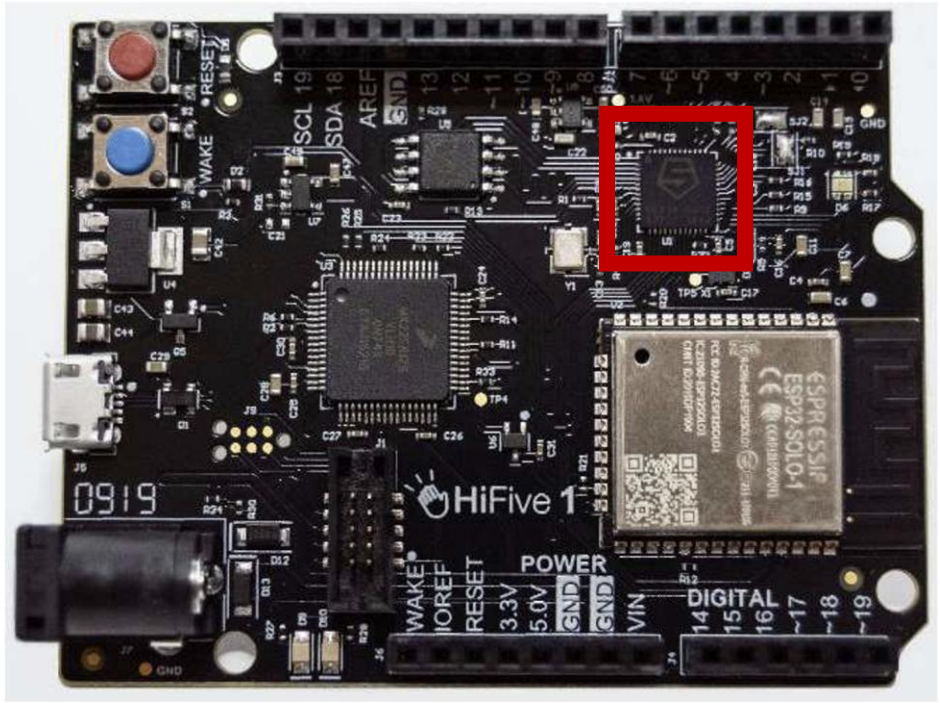
- Design a CPU to execute all RV64I instructions

RV64i Base Integer Instructions

Opcode	Instruction	Fmt	Example		Description	Notes
li	load immediate	*	li	a0, 2	addi a0, zero, 2	*pseudo*
la	load address	*	la	a0, symbol	a0 = symbol	*pseudo*, 2 instr
add	add	R	add	a0, a1, a2	a0 = a1 + a2	
sub	subtract	R	sub	a0, a1, a2	a0 = a1 - a2	
xor	bitwise exclusive or	R	xor	a0, a1, a2	a0 = a1 ^ a2	
or	bitwise or	R	or	a0, a1, a2	a0 = a1 | a2	
and	bitwise and	R	and	a0, a1, a2	a0 = a1 & a2	
sll	shift left logical	R	sll	a0, a1, a2	a0 = a1 << a2	
srl	shift right logical	R	srl	a0, a1, a2	a0 = a1 >> a2	
sra	shift right arith*	R	sra	a0, a1, a2	a0 = a1 >> a2	sign-extends
slt	set less than	R	slt	a0, a1, a2	a0 = (a1 < a2) ? 1 : 0	
sltu	set less than (u)	R	sltu	a0, a1, a2	a0 = (a1 < a2) ? 1 : 0	unsigned
addi	add immediate	I	addi	a0, a1, 2	a0 = a1 + 2	
xori	xor immediate	I	xori	a0, a1, 2	a0 = a1 ^ 2	
ori	or immediate	I	ori	a0, a1, 2	a0 = a1 | 2	
andi	and immediate	I	andi	a0, a1, 2	a0 = a1 & 2	
slli	shift left logical imm	I	slli	a0, a1, 2	a0 = a1 << 2	
srli	shift right logical imm	I	srli	a0, a1, 2	a0 = a1 >> 2	
srai	shift right arith imm	I	srai	a0, a1, 2	a0 = a1 >> 2	sign-extends
slti	set less than imm	I	slti	a0, a1, 2	a0 = (a1 < 2) ? 1 : 0	
sltiu	set less than imm (u)	I	sltiu	a0, a1, 2	a0 = (a1 < 2) ? 1 : 0	unsigned
mv	move (copy)	*	mv	a0, a1	addi a0, a1, 0	*pseudo*
neg	2s-complement negation	*	neg	a0, a1	sub a0, zero, a1	*pseudo*
not	bitwise not	*	not	a0, a1	xori a0, a1, -1	*pseudo*
lb	load byte	I	lb	a0, 1(a1)	a0 = M[a1+1] (8 bits)	
lh	load half	I	lh	a0, 2(a1)	a0 = M[a1+2] (16 bits)	
lw	load word	I	lw	a0, 4(a1)	a0 = M[a1+4] (32 bits)	
ld	load double word	I	ld	a0, 8(a1)	a0 = M[a1+8] (64 bits)	
lbu	load byte (u)	I	lbu	a0, 1(a1)	a0 = M[a1+1] (8 bits)	zero-extends
lhu	load half (u)	I	lhu	a0, 2(a1)	a0 = M[a1+2] (16 bits)	zero-extends
lwu	load word (u)	I	lwu	a0, 4(a1)	a0 = M[a1+4] (32 bits)	zero-extends
l{b|h|w|d}	load global	*	ld	a0, symbol	a0 = M[symbol]	*pseudo*, 2 instr
sb	store byte	S	sb	a0, 1(a1)	M[a1+1] = a0 (8 bits)	
sh	store half	S	sh	a0, 2(a1)	M[a1+2] = a0 (16 bits)	
sw	store word	S	sw	a0, 4(a1)	M[a1+4] = a0 (32 bits)	
sd	store double word	S	sd	a0, 8(a1)	M[a1+8] = a0 (64 bits)	
s{b|h|w|d}	store global	*	sd	a0, symbol, t0	M[symbol] = a0 (uses t0)	*pseudo*, 2 instr
beq	branch if =	B	beq	a0, a1, 2b	if (a0 == a1) goto 2b	
bne	branch if ≠	B	bne	a0, a1, 2f	if (a0 != a1) goto 2f	

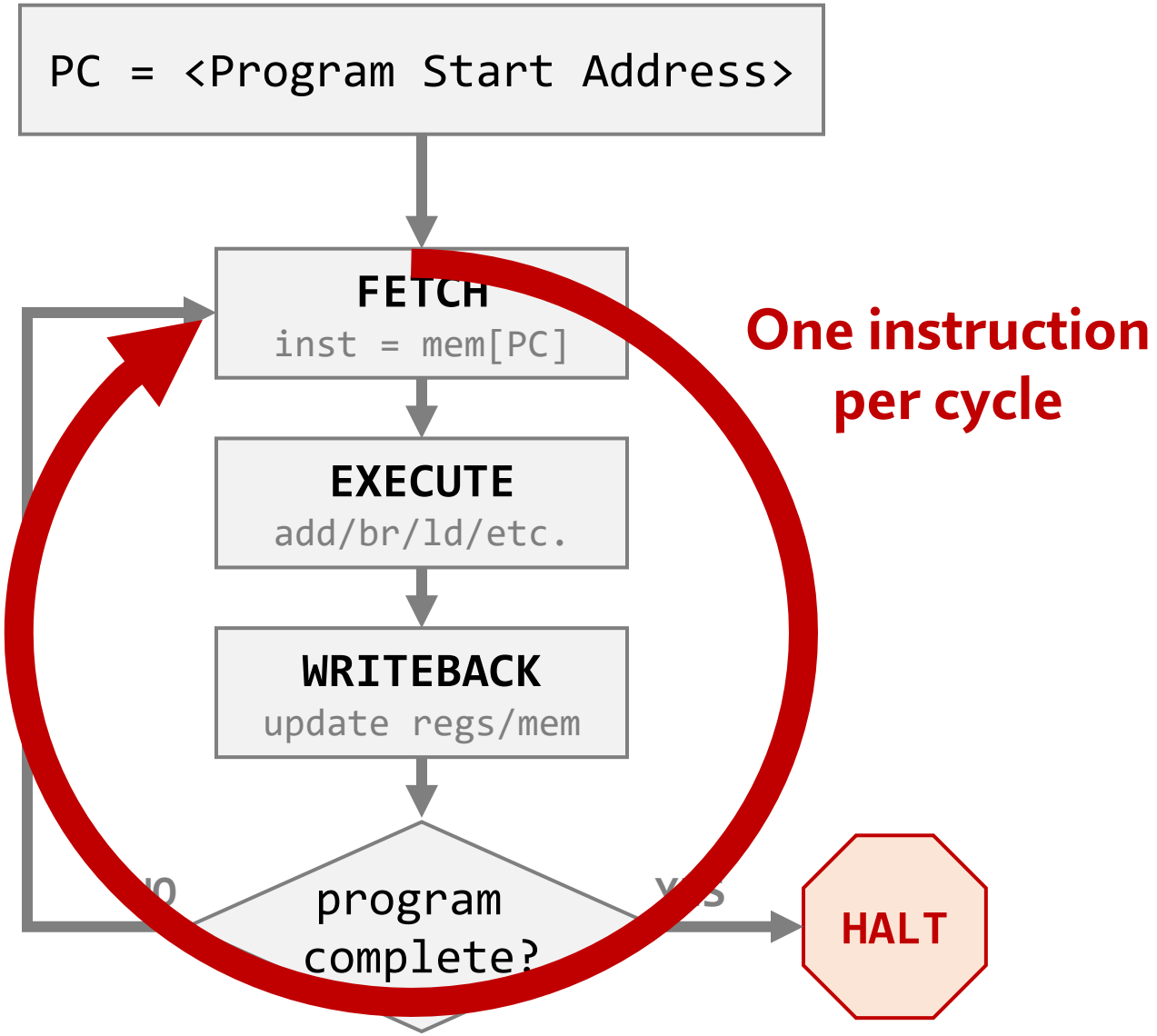
https://www.cs.utahtech.edu/cs/2810/riscv-card.pdf

HiFive 1 Rev B01



SiFive, "HiFive1 Rev B Schematics," 2021.

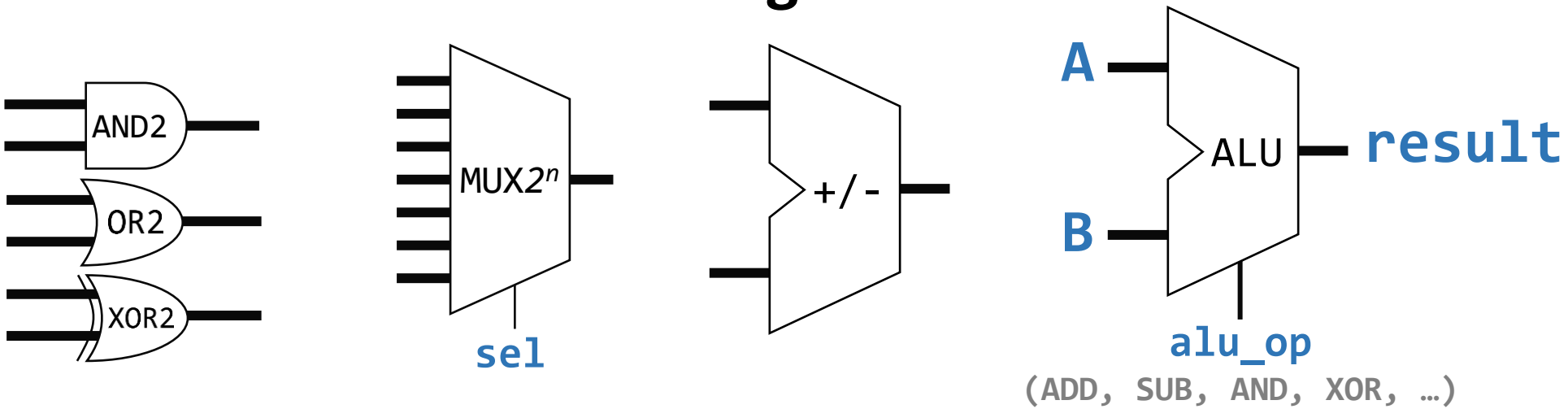
General CPU Control Flow



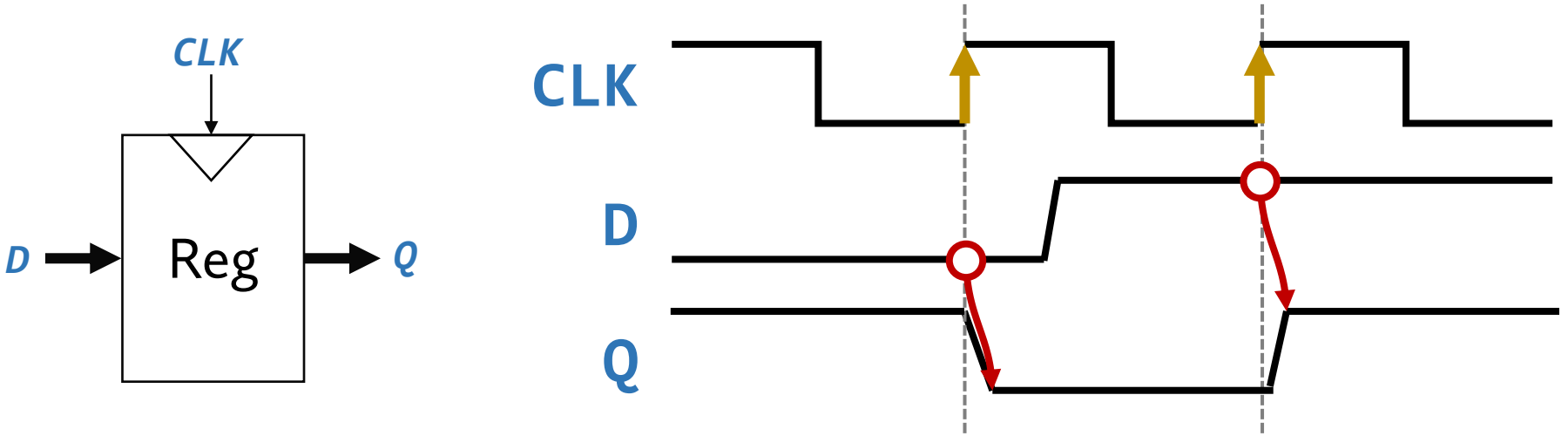
One instruction per cycle

Our Building Blocks

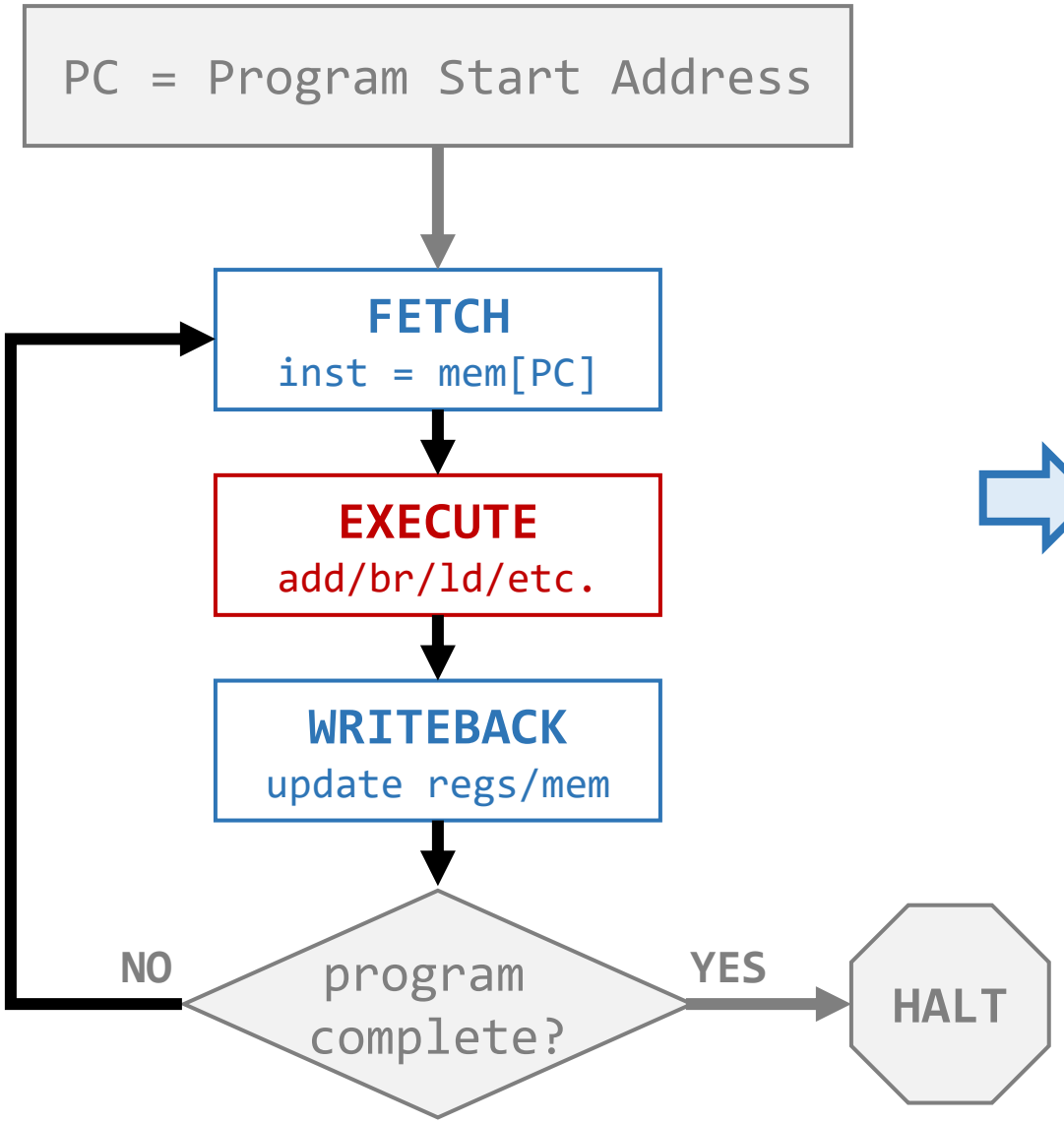
Combinational Logic Blocks

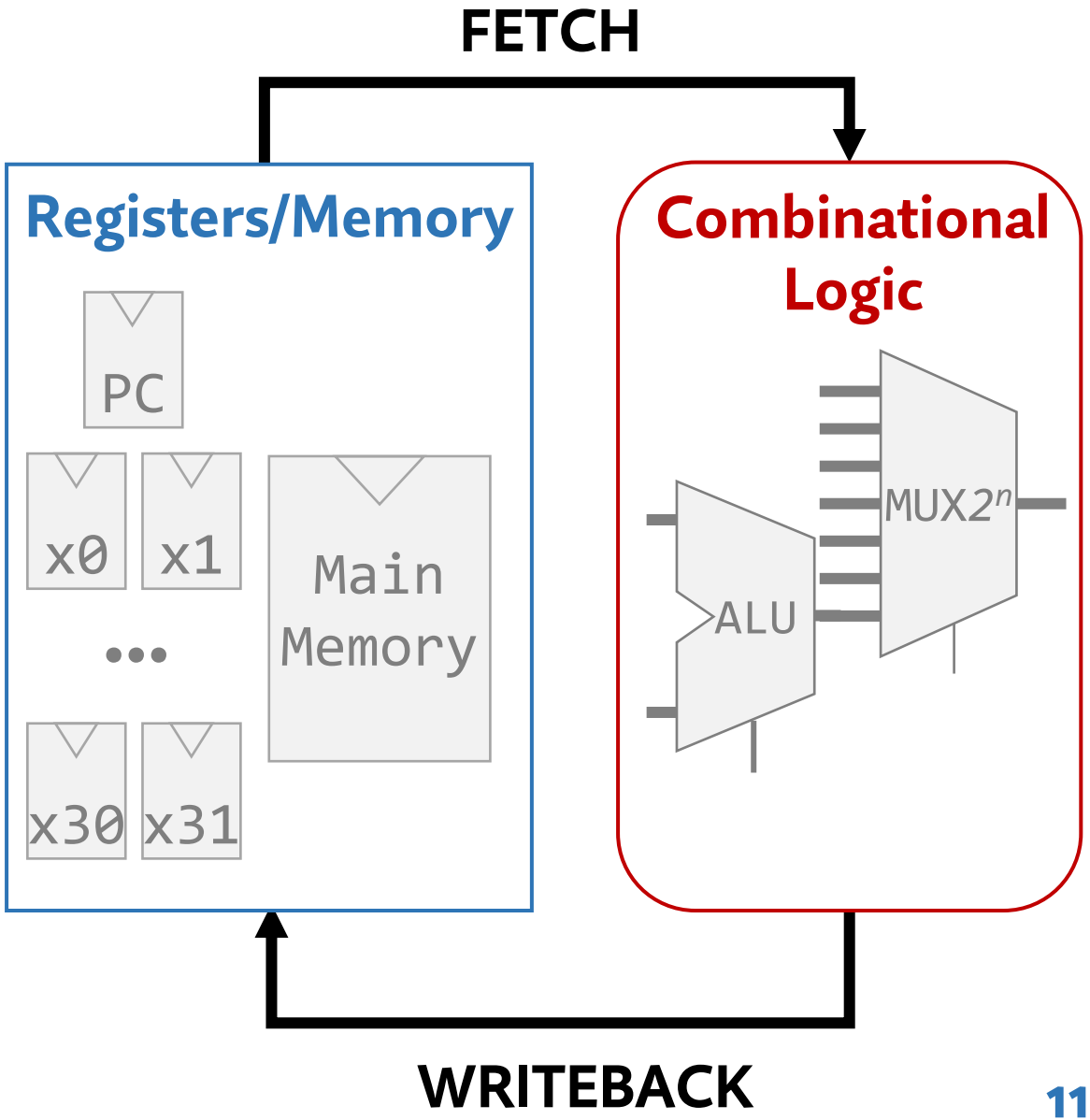


Registers

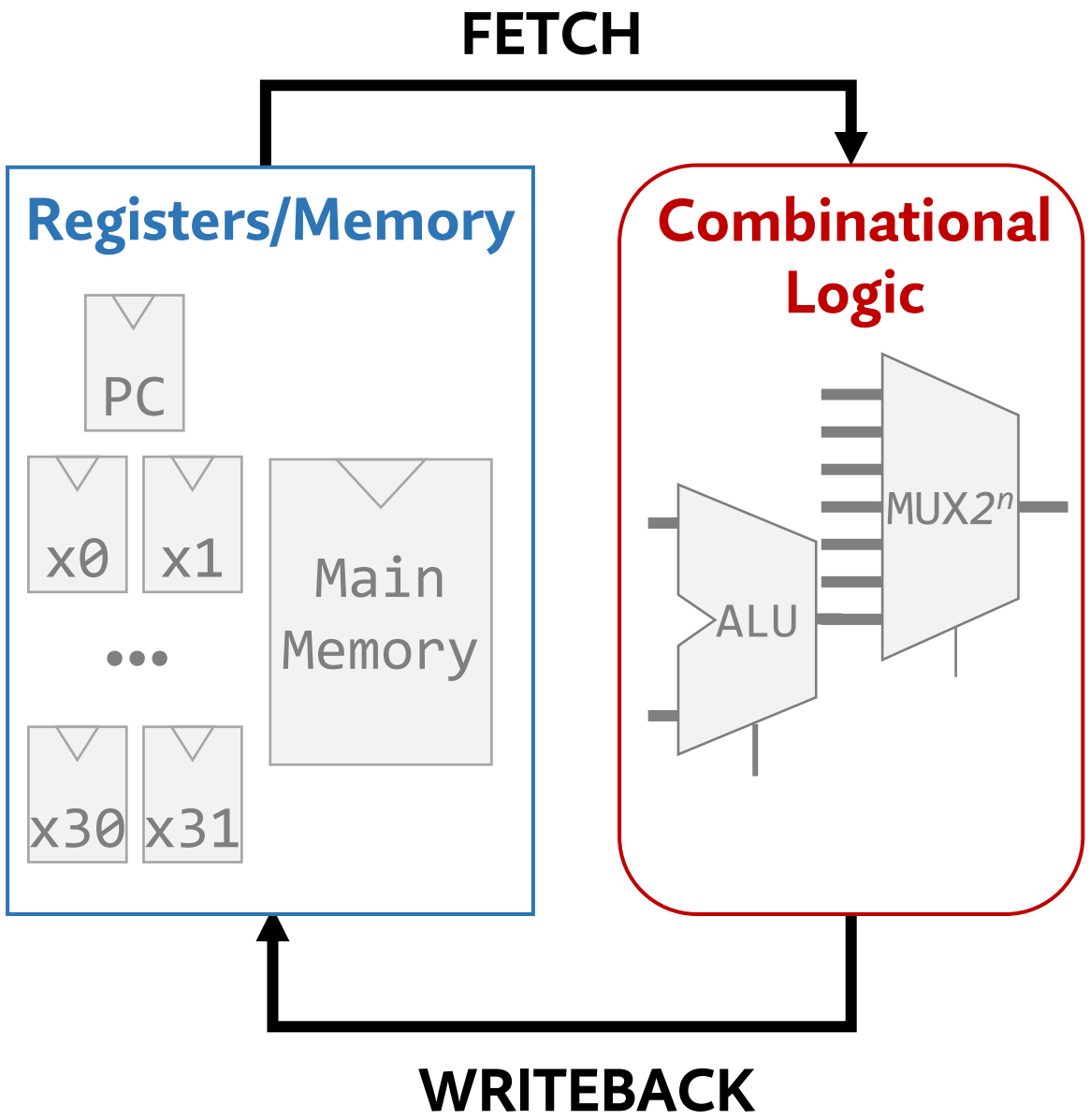


From Abstraction to Logic Circuits

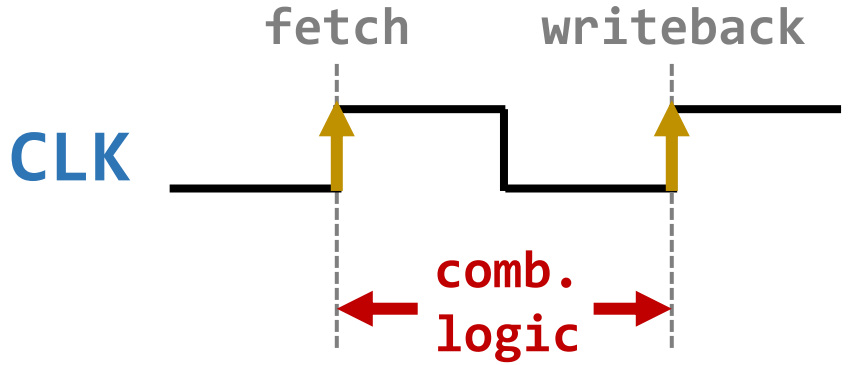




One Instruction Per Clock Cycle



Executes **any** RV64I instruction within **one clock cycle**



$$freq = \frac{1}{cycle\ time}$$

Example: `4 GHz = 4 billion cycles/second`

Cycle time: `0.25 ns`

4 Billion Instructions per Second

- How long would it take for you to calculate **four billion** 64-bit ADDs?

Pen + Paper

$$4 \text{ billion adds} \times \frac{60 \text{ seconds}}{\text{add}} \times \frac{1 \text{ hour}}{3600 \text{ seconds}} \times \frac{1 \text{ year}}{8760 \text{ hours}} = 7610.35 \text{ years}$$

4 GHz Single-Cycle CPU

1 second*

**lots of assumptions (e.g., no memory accesses)*

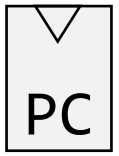
Agenda

- Single-Cycle CPU
  - **State Elements**
- Five-Stage Execution Model
  - Implementing an ADD
  - R-TYPE Instructions
  - Any Instruction

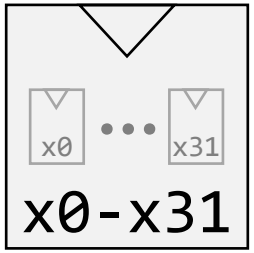
Three State Elements

- Instructions modify the system state

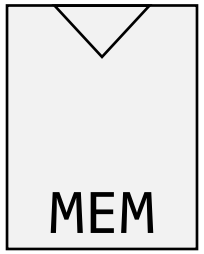
Register

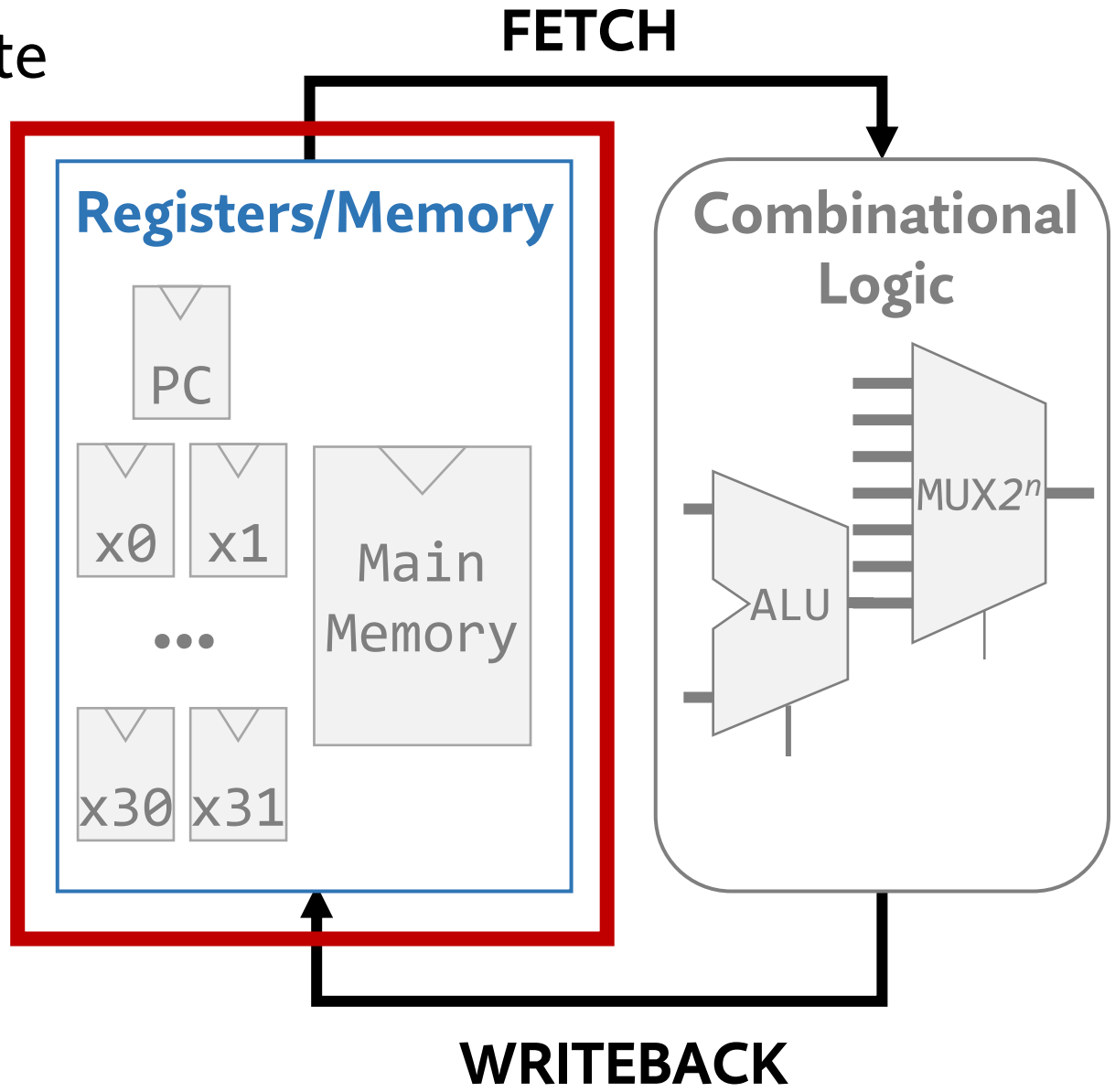


Register File



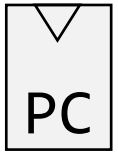
Memory





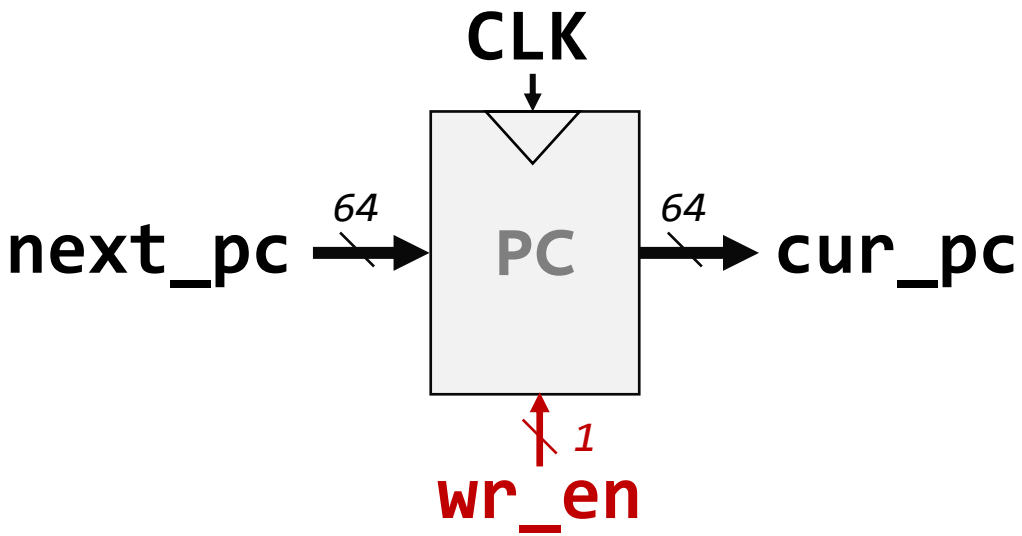
State Element 1/3: Program Counter

Register



Register File

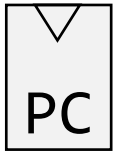
Memory



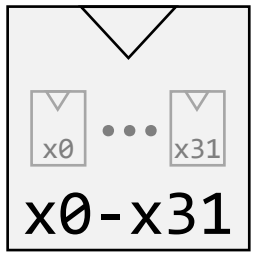
- **`cur_pc`** is stable **throughout** the clock cycle
- **`if(wr_en)`**
  `cur_pc` becomes **`next_pc`** at the **end of the cycle**

State Element 2/3: Register File

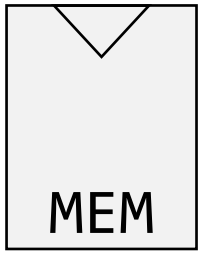
Register

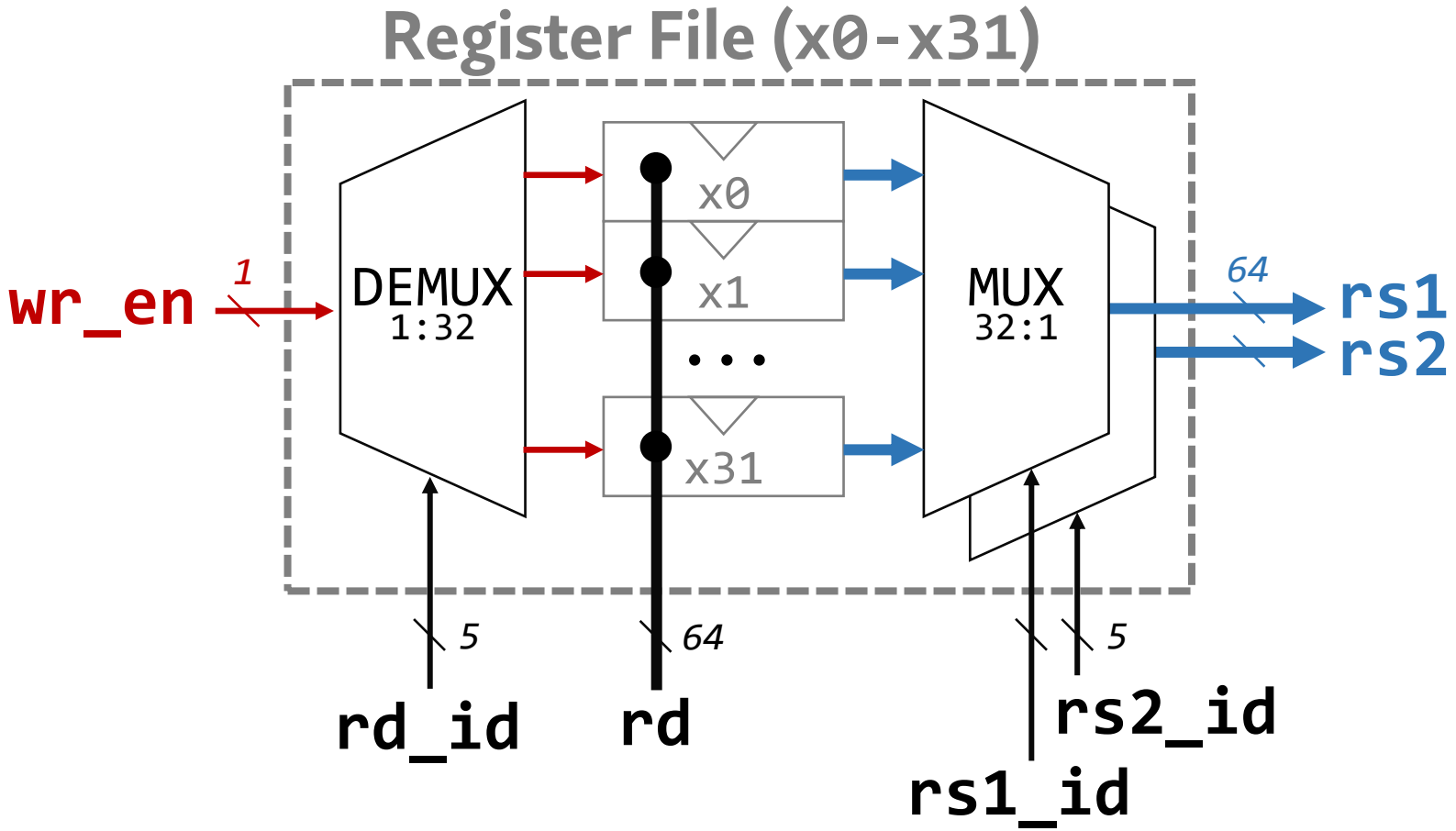


Register File



Memory



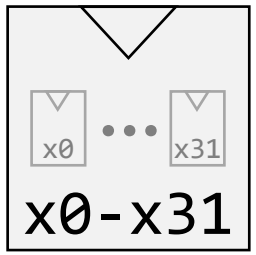


- Combines 32 registers into one logic block
  - **`rd`**: one write input
  - **`rs1/rs2`**: two read outputs

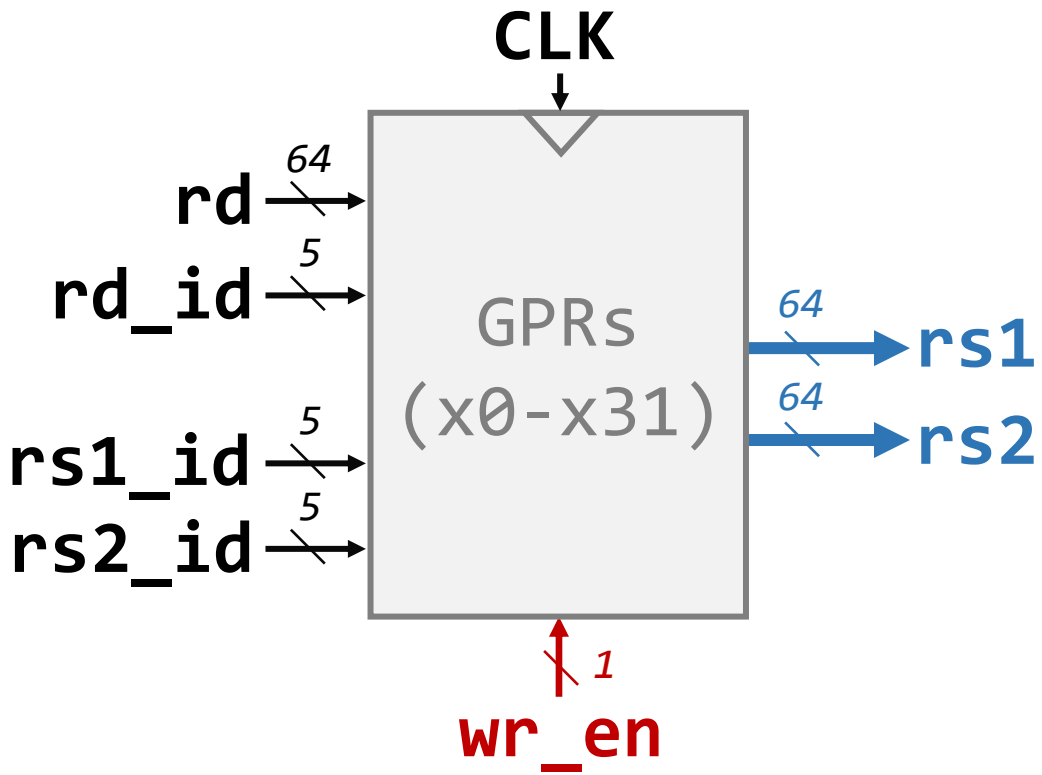
State Element 2/3: Register File (Abstract)

Register

Register File



Memory



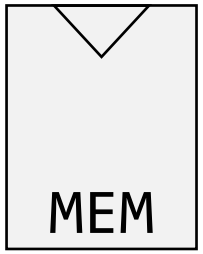
- Combines 32 registers into one logic block
  - **`rd`**: one write input
  - **`rs1/rs2`**: two read outputs

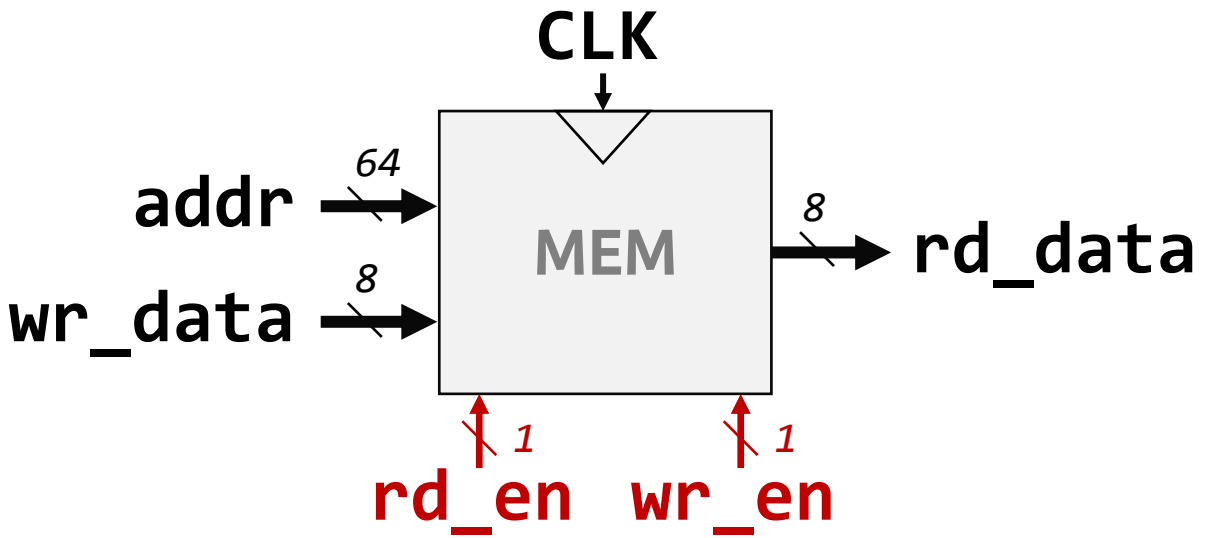
State Element 3/3: Main Memory

Register

Register File

Memory

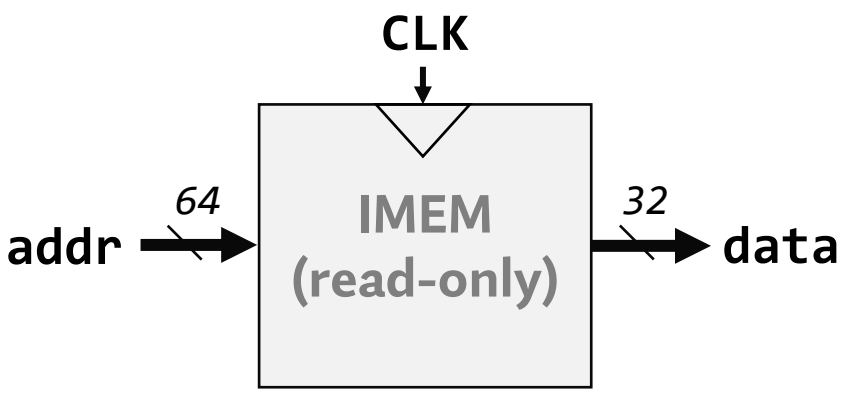


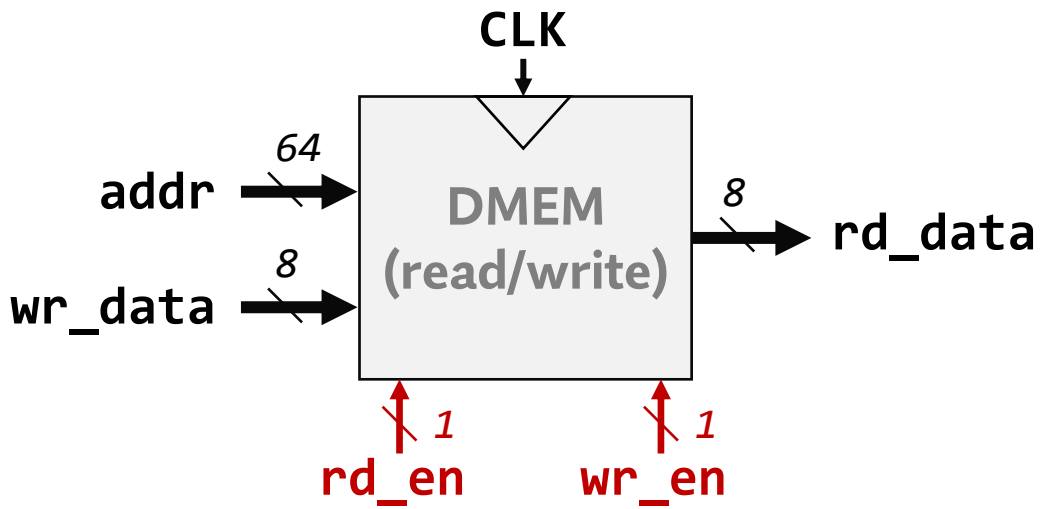


- Memory is **"magic"**
- Assume:
  - Byte addressable (8-bit read/write interface)
  - 64-bit address (2^{64} unique byte locations)

"Two" Memories: Instructions and Data

- We draw main memory as split into two parts (**just notation!**)



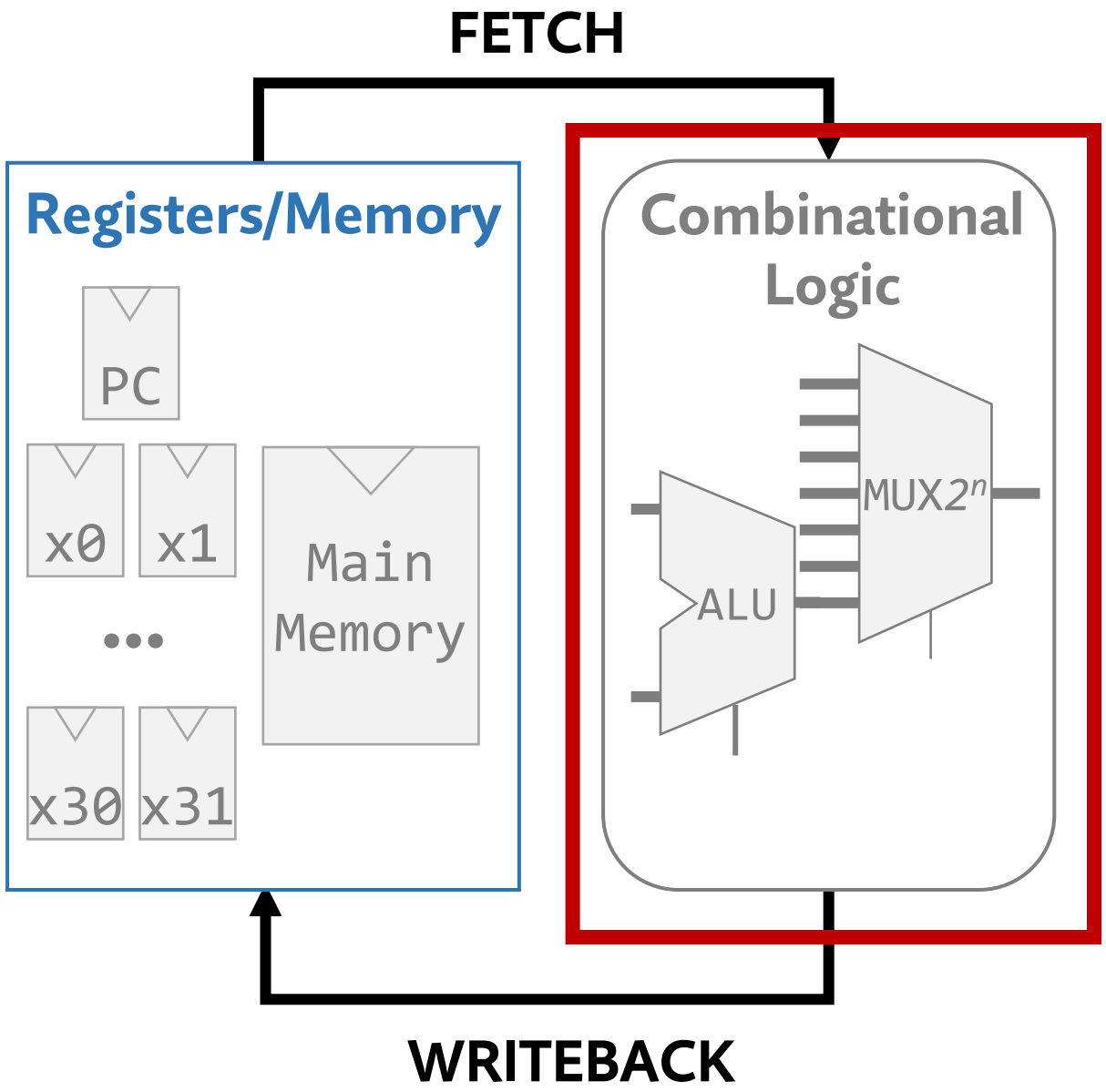


- There's still only one **actual main memory**
  - IMEM/DMEM are placeholders for **caches** (more on that later)

Agenda

- Single-Cycle CPU
  - State Elements
- **Five-Stage Execution Model**
  - Implementing an ADD
  - R-TYPE Instructions
  - Any Instruction

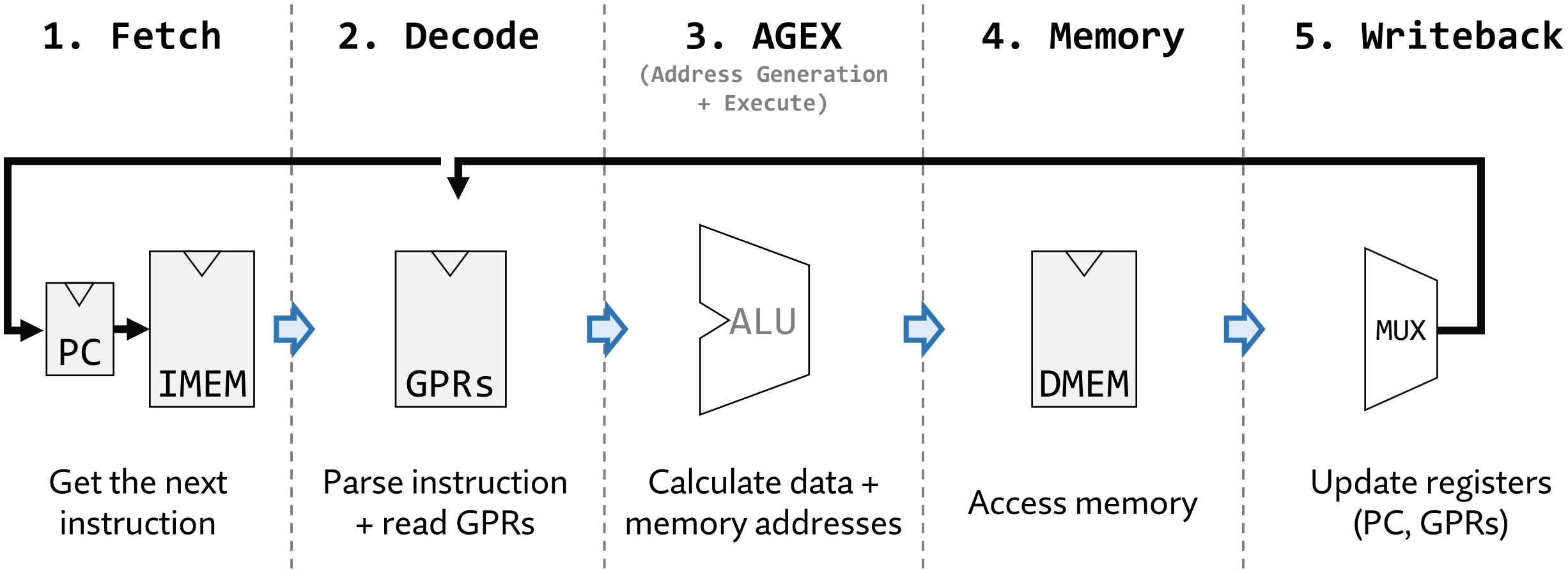
Implementing the Combinational Logic



Problem: hard to design and build as one giant blob

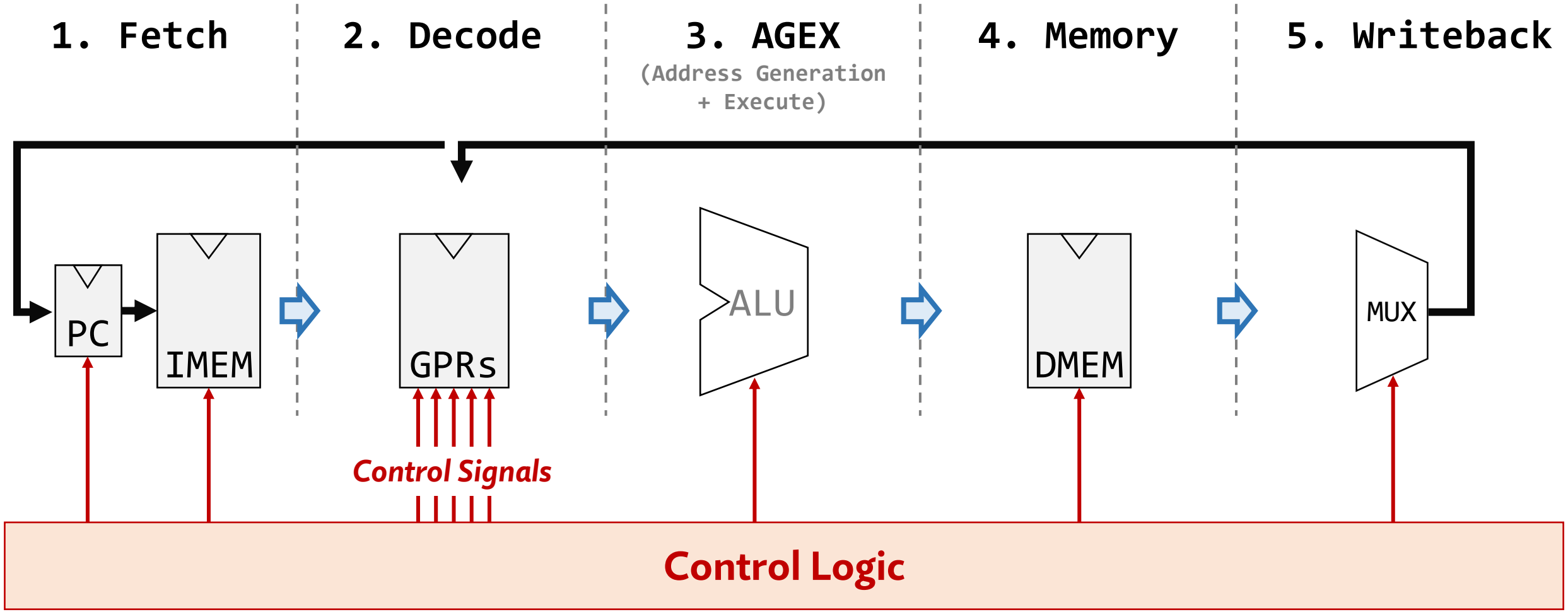
Solution: split it into smaller "stages" and connect them together

Five-Stage CPU: Datapath



Datapath: Should have everything we need to execute any instruction

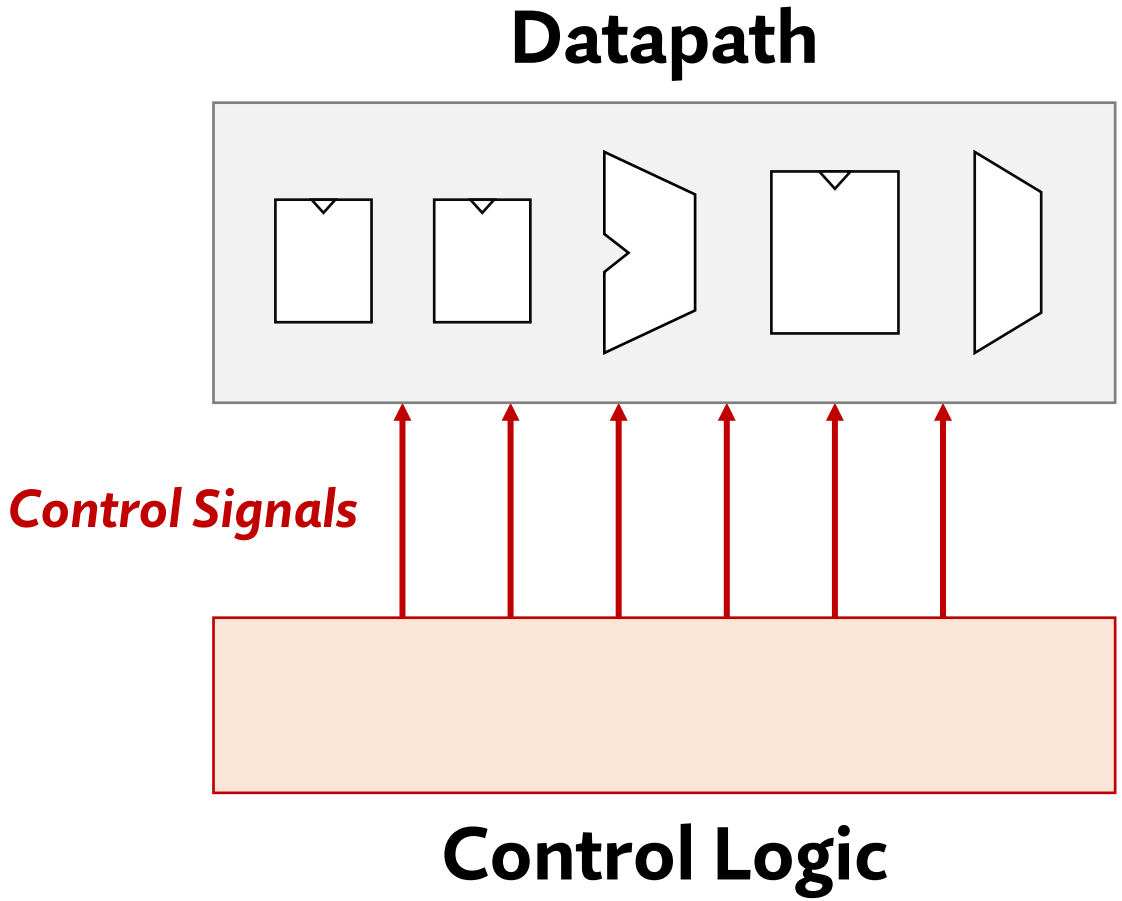
Five-Stage CPU: Control Logic



- Control logic decides what to do **for a specific instruction** (e.g., add)

Summary: General CPU Architecture

- **Datapath** provides logic to perform *any* RISC-V operation



- **Control logic** orchestrates the datapath for a given instruction

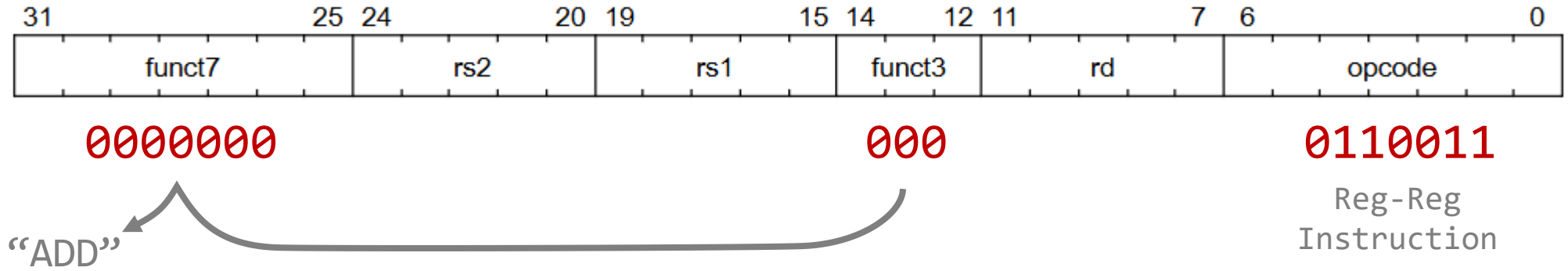
Agenda

- Single-Cycle CPU
  - State Elements
- Five-Stage Execution Model
  - **Implementing an ADD**
  - R-TYPE Instructions
  - Any Instruction

Starting with an add

- **Approach:** start simple

```
add rd, rs1, rs2
```

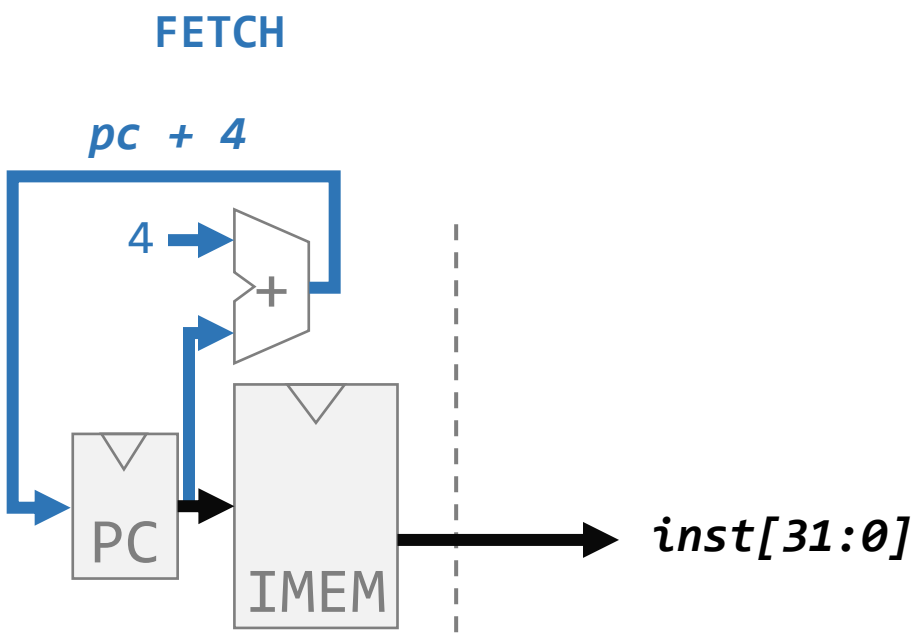


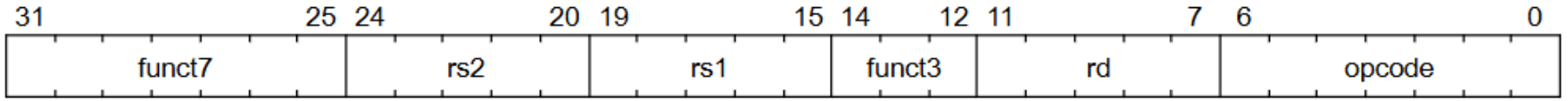
State Element	Action
PC	`PC += 4`
GPRs	`R[rd] = R[rs1] + R[rs2]`
DMEM	-

Just need the **control signals** that make all this happen

Datapath for add

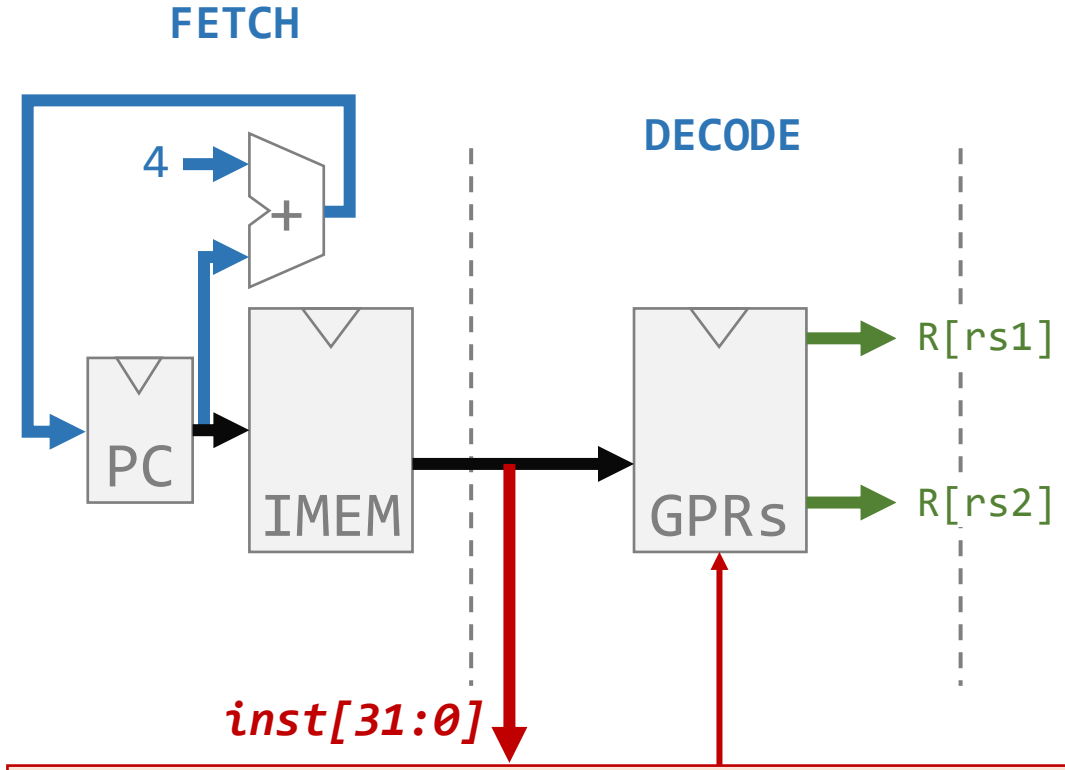
State Element	Action
PC	PC += 4
GPRs	R[rd] = R[rs1] + R[rs2]
DMEM	-

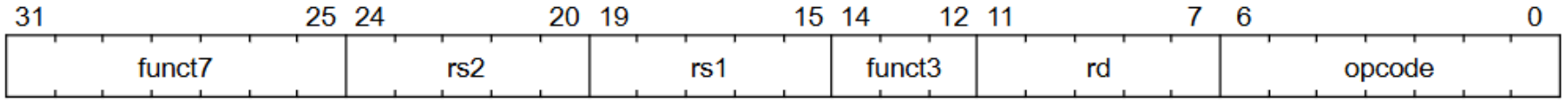




Datapath for add

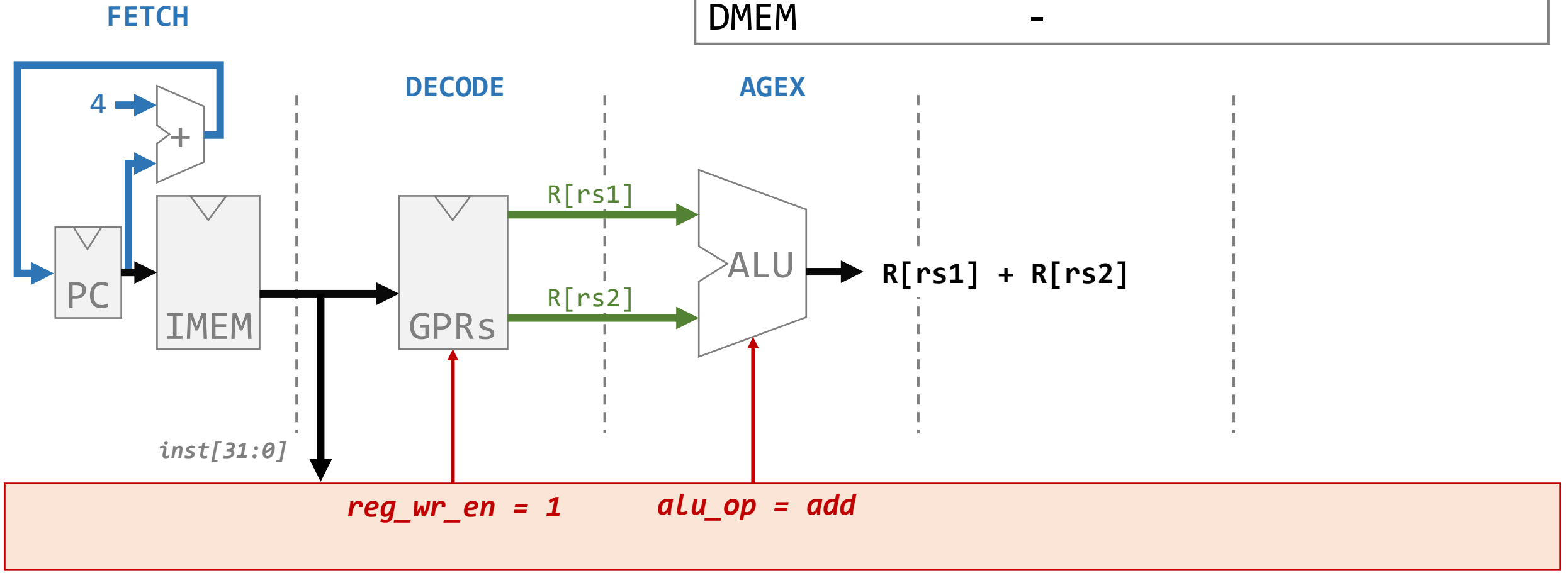
State Element	Action
PC	PC += 4
GPRs	R[rd] = R[rs1] + R[rs2]
DMEM	-

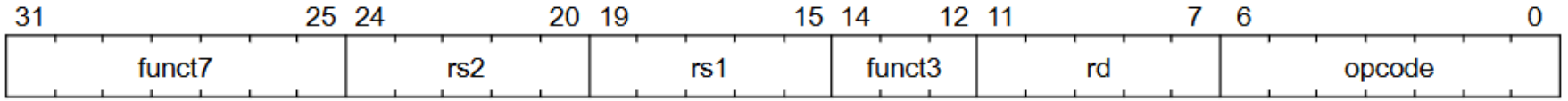




Datapath for add

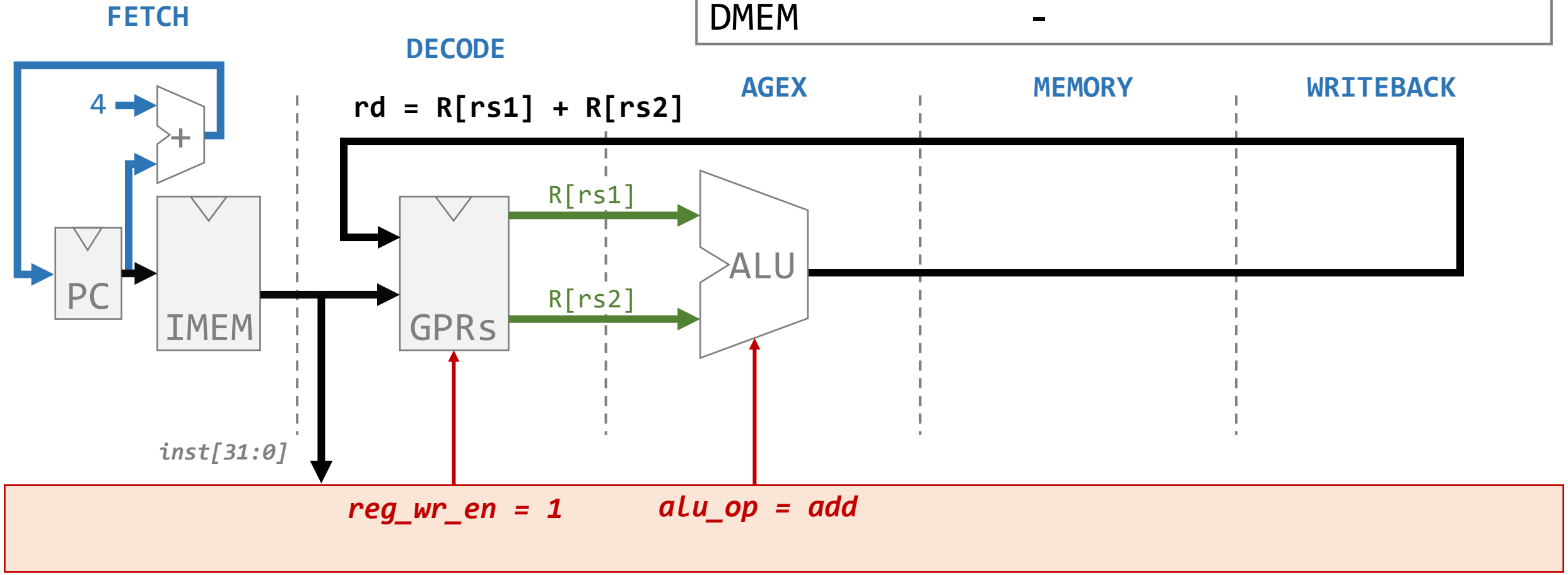
State Element	Action
PC	PC += 4
GPRs	R[rd] = R[rs1] + R[rs2]
DMEM	-

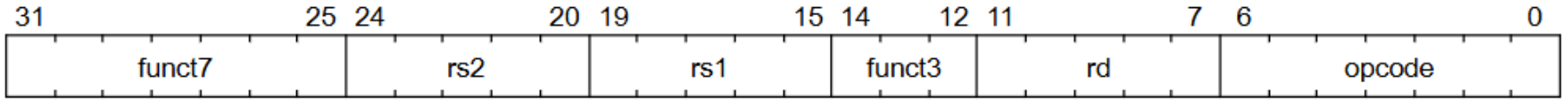




Datapath for add

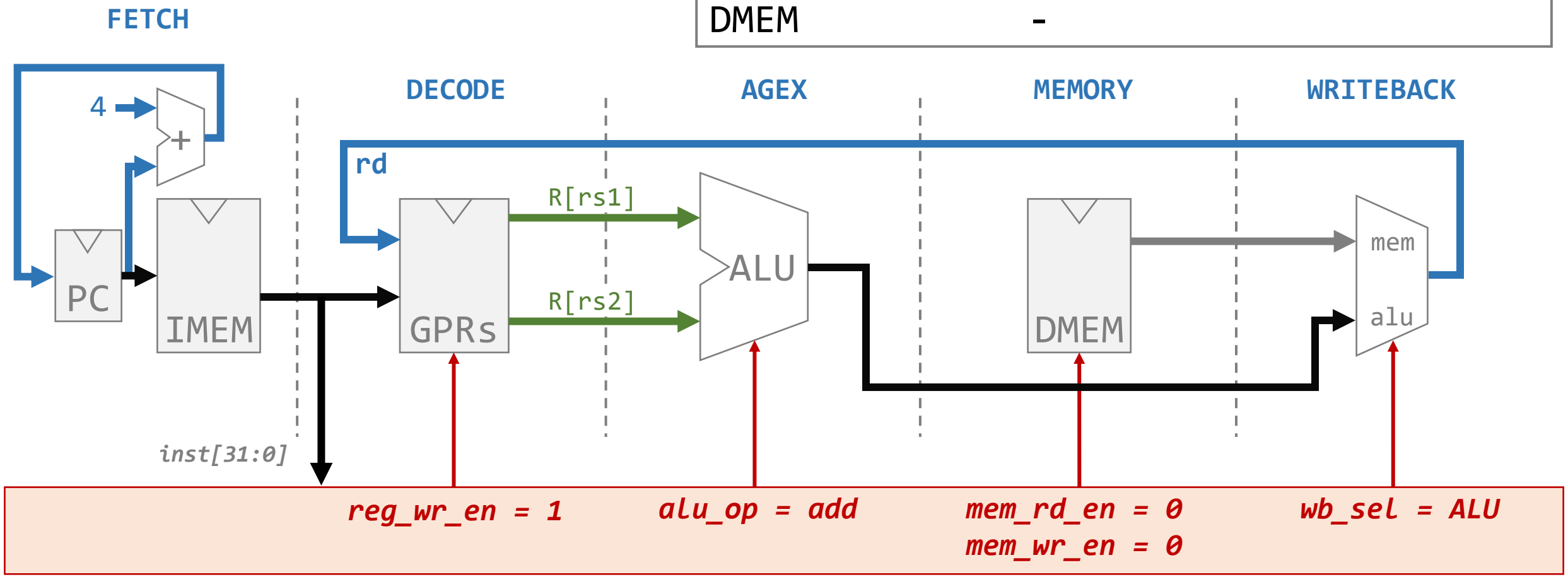
State Element	Action
PC	PC += 4
GPRs	R[rd] = R[rs1] + R[rs2]
DMEM	-

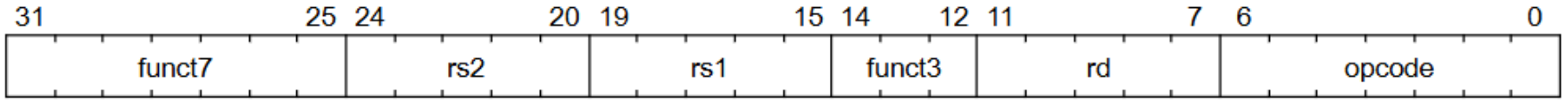




Datapath for add

State Element	Action
PC	PC += 4
GPRs	R[rd] = R[rs1] + R[rs2]
DMEM	-





Agenda

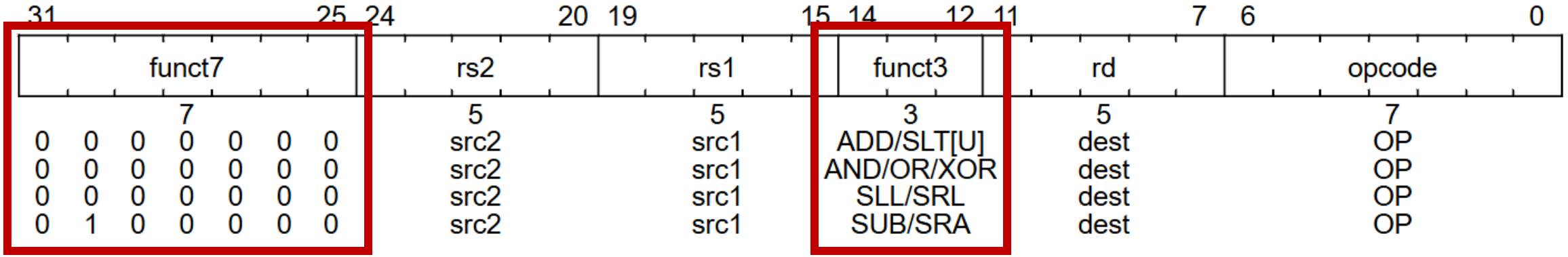
- Single-Cycle CPU
  - State Elements
- Five-Stage Execution Model
  - Implementing an ADD
  - **R-TYPE Instructions**
  - Any Instruction

Extending the Datapath

- Would like to support **all RISC-V instructions**
- Much easier if we **group similar instructions**
  - Luckily, the ISA designers already did this for us

R-TYPE (reg-reg)

`<op> rd, rs1, rs2`

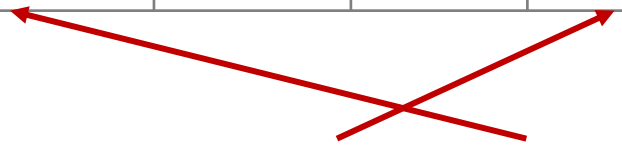


- Control logic:
  1. Figures out this is an R-Type instruction
  2. Parses **`funct3`** and **`funct7`** to figure out the ALU operation

Supporting All R-Type Instructions

```
<op> rd, rs1, rs2
```

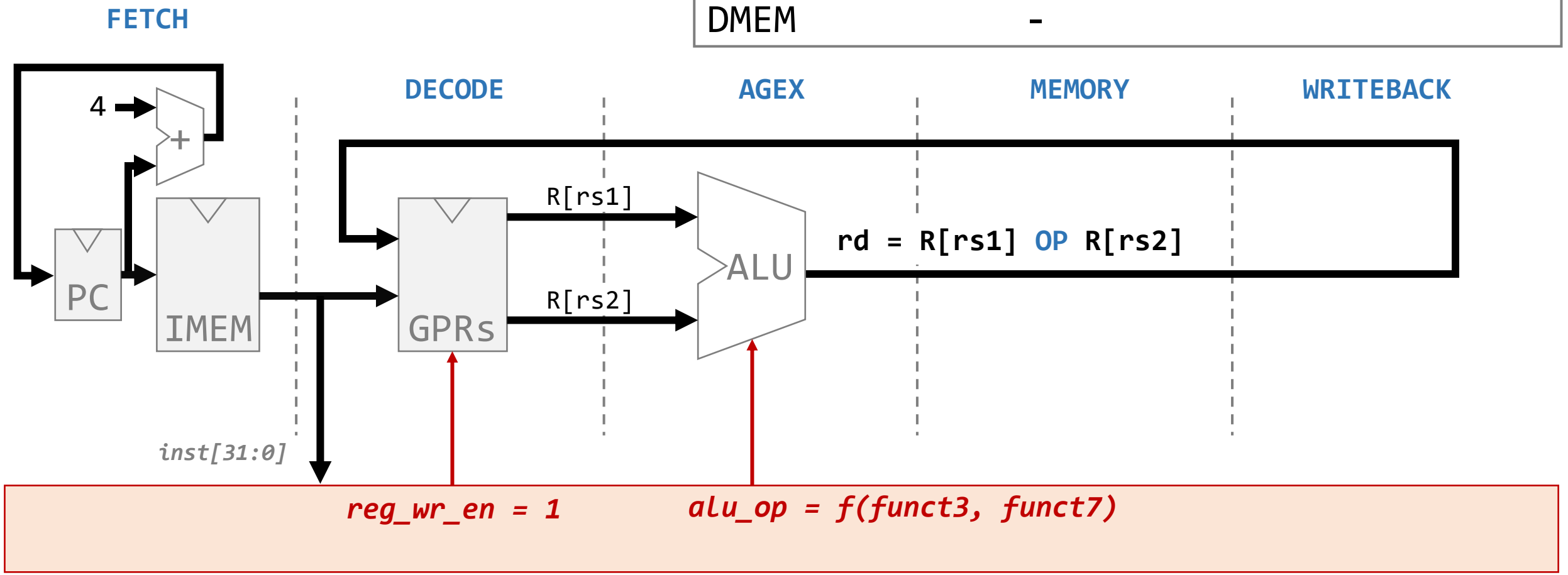
funct7			funct3		opcode	<op>
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

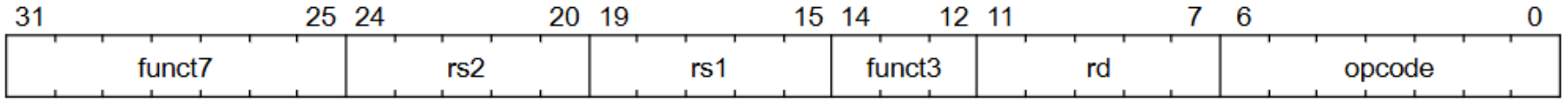


```
alu_op = f(funct3, funct7)
```

R-TYPE Datapath

State Element	Action
PC	PC += 4
GPRs	R[rd] = R[rs1] OP R[rs2]
DMEM	-

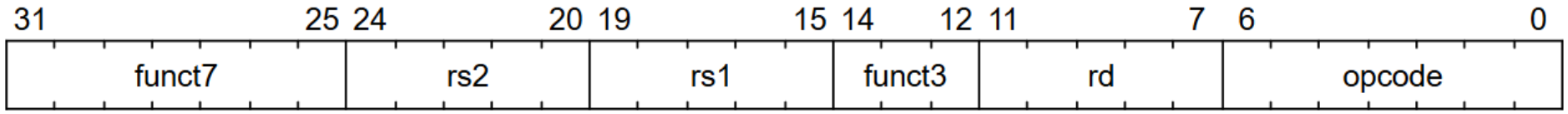




Agenda

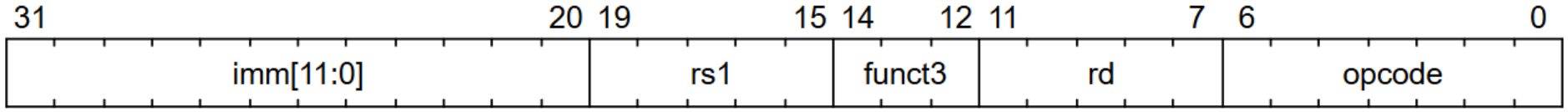
- Single-Cycle CPU
  - State Elements
- Five-Stage Execution Model
  - Implementing an ADD
  - R-TYPE Instructions
  - **Any Instruction**

6 Types of RISC-V Instructions



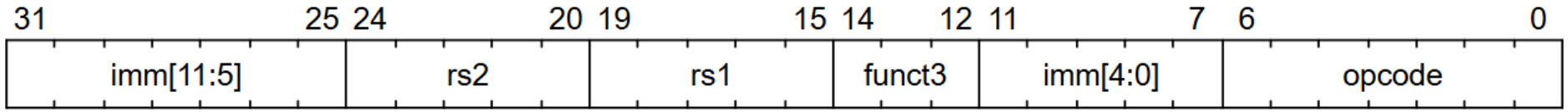
R-Type

```
add, sub, slt(u),
xor, or, and,
sll, srl, sra
```



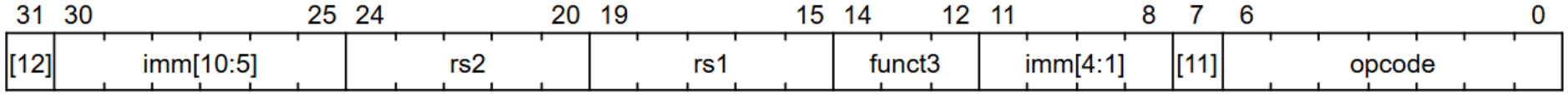
I-Type

```
addi, subi, slt(u)i,
xori, ori, andi, slli,
srli, srai, lb(u),
lh(u), lw(u), ld, jalr
```



S-Type

```
sb, sh, sw, sd
```

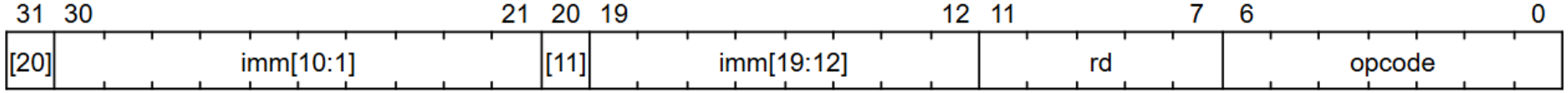


B-Type

```
b<cond>
```

U-Type

```
lui, auipc
```



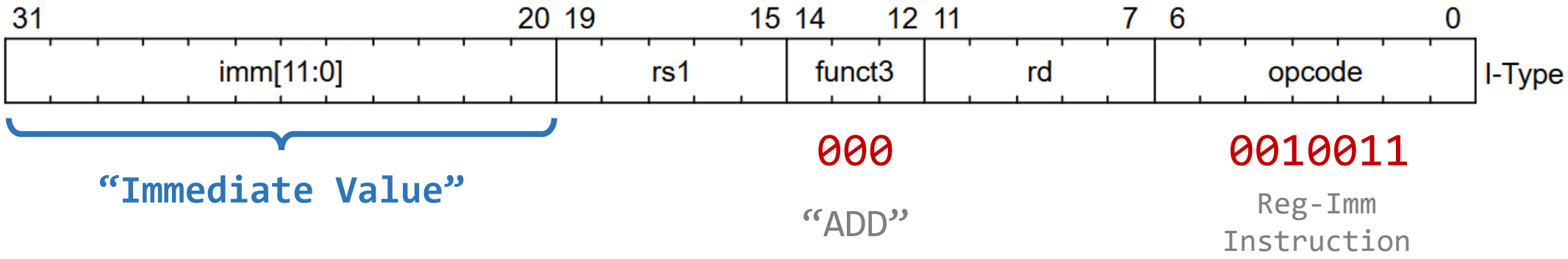
J-Type

```
jal
```

ADDI Instruction (I-Type)

- Similar to ADD, but **with an immediate value** instead

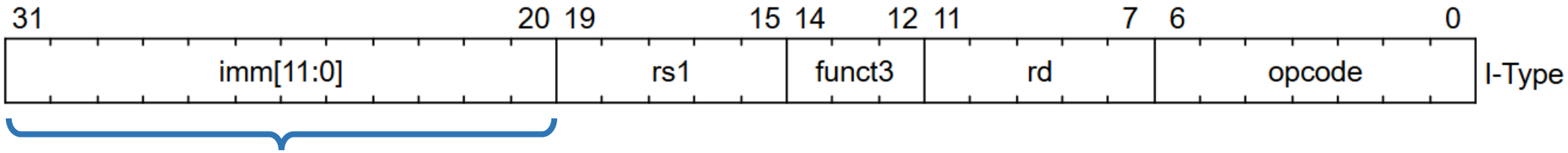
```
addi rd, rs1, imm
```

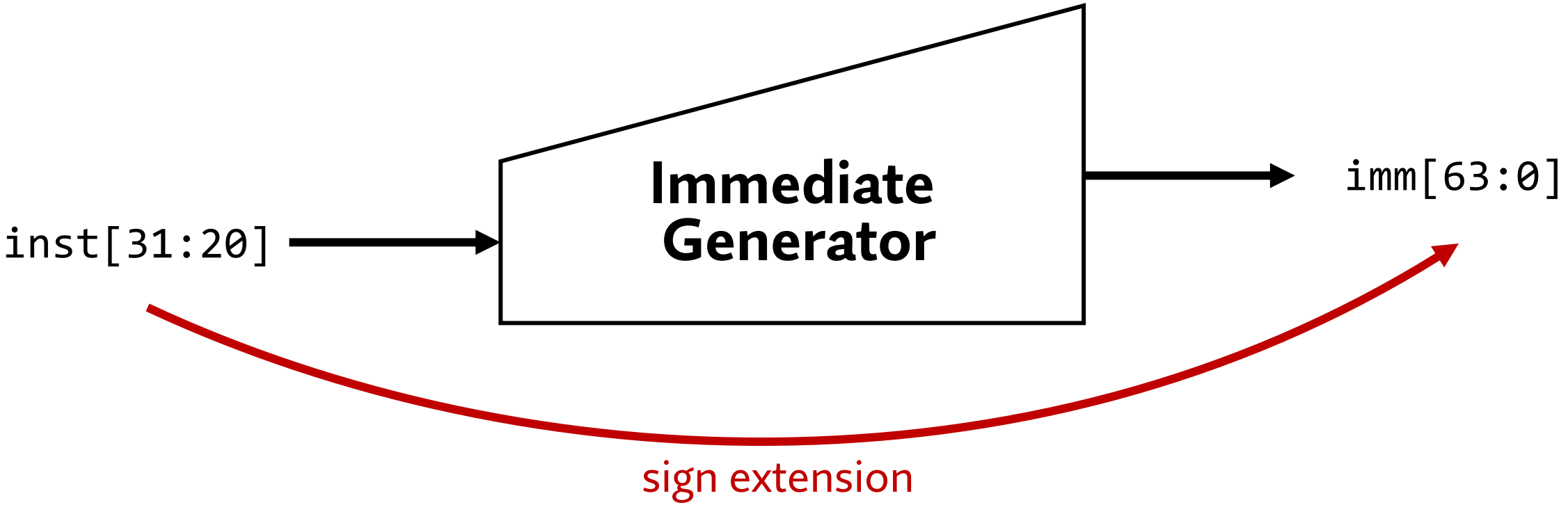


State Element	Action
PC	PC += 4
GPRs	R[rd] = R[rs1] + **sext(imm[11:0])**
DMEM	-

Need some logic block to generate **imm**

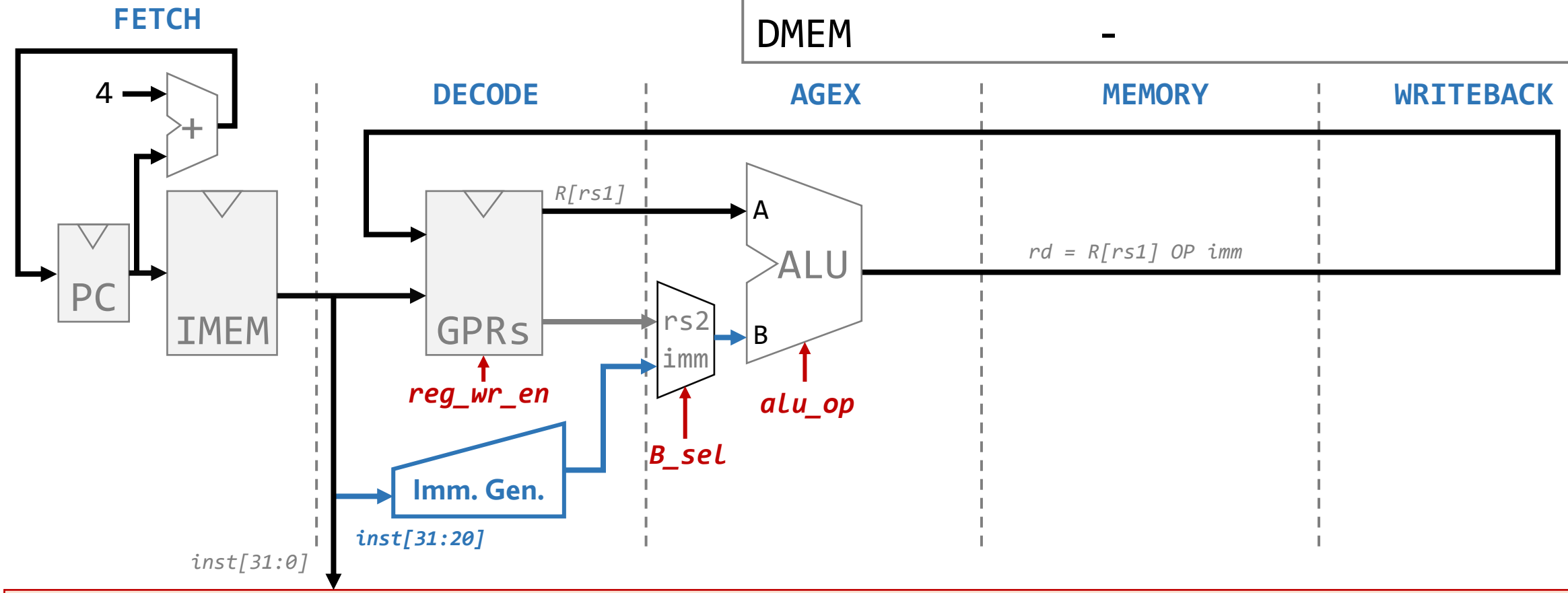
Extracting an Immediate Value

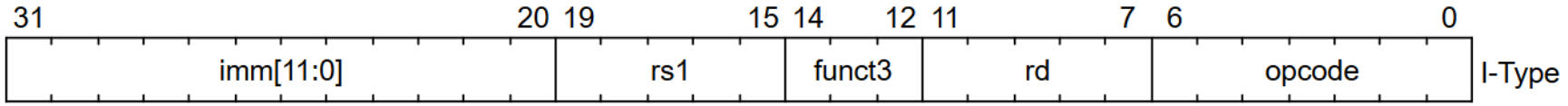




I-TYPE Datapath

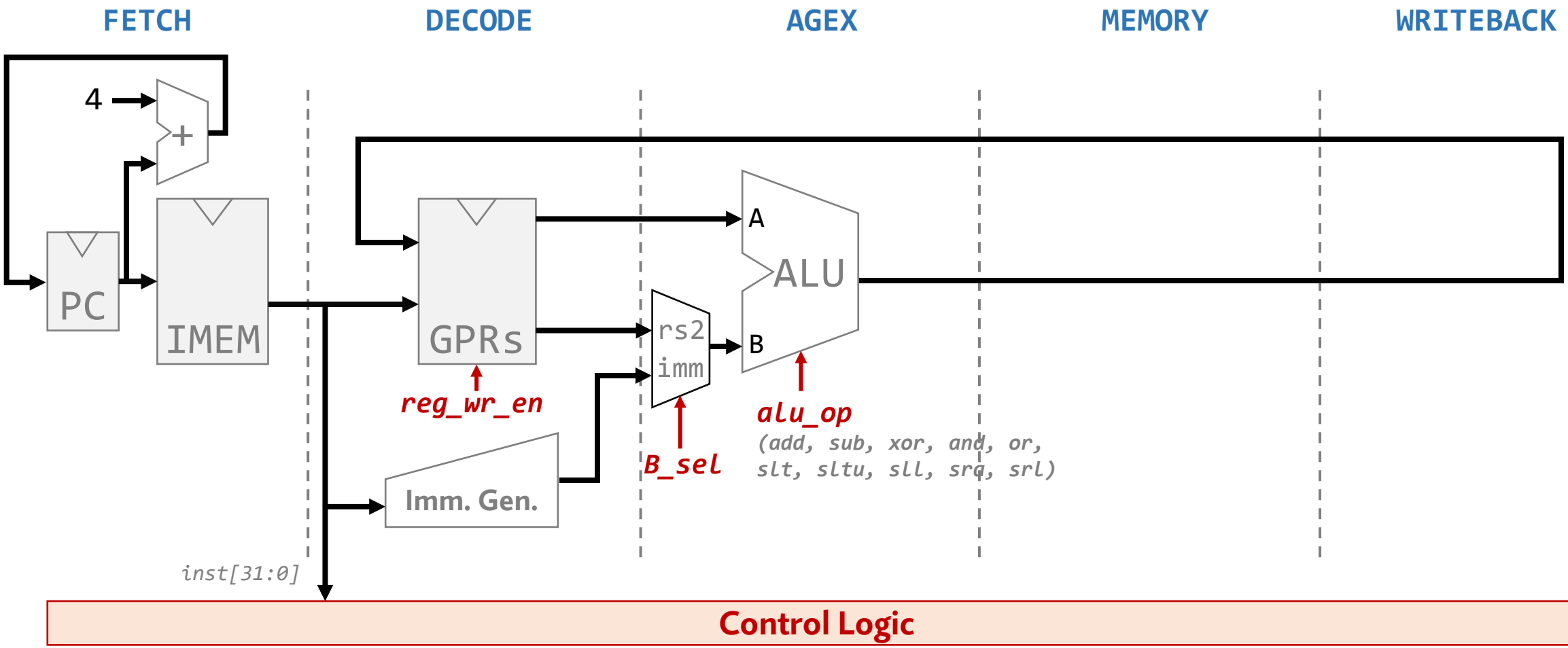
State Element	Action
PC	PC += 4
GPRs	R[rd] = R[rs1] OP imm
DMEM	-





Summary: Arithmetic/Logical Datapath

- **Datapath** propagates information
- **Control logic** chooses what to use vs. ignore



CS 211: Intro to Computer Architecture

12.1: Single-Cycle RV64I CPU - Arithmetic/Logic

Minesh Patel

Spring 2025 – Tuesday 15 April